An Experiment in Interoperable Cryptographic Protocol Implementation Using Automatic Code Generation

Alfredo Pironti, Riccardo Sisto

Politecnico di Torino

Dip. di Automatica e Informatica

c.so Duca degli Abruzzi 24, I-10129 Torino (Italy)

e-mail:{alfredo.pironti, riccardo.sisto}@polito.it

phone: +390115647073 fax: +390115647099

Abstract

spi2java is a tool that enables semi-automatic generation of cryptographic protocol implementations, starting from verified formal models. This paper shows how the last version of spi2java has been enhanced in order to enable interoperability of the generated implementations. The new features that have been added to spi2java are reported here. A case study on the SSH Transport Layer Protocol, along with some experiments and measures on the generated code, is also provided. The case study shows, with facts, that reliable and interoperable implementations of standard security protocols can indeed be obtained by using a code generation tool like spi2java.

1 Introduction

Given a security protocol specification, it is unsafe to manually write the code that implements the given specification, because there is no assurance that the written code correctly adheres to the specification, which can lead to the introduction of severe security flaws, not present in the specification, but added by wrong coding.

By using techniques such as testing or code reviews, it can be assured that the program is correctly working only for a limited number of scenarios, but it cannot be verified that it will behave as specified under all circumstances.

Formal methods can help to tackle the above problem. However, while a lot of progress has been made on the use of formal methods to verify the correctness of cryptographic protocol specifications, only a few research works have addressed the problem of ensuring that the protocol implementation, written in a programming language, correctly implements the protocol specification. This paper focuses on methods based on automatic code generation from formal specifications. All such methods start from a high-level, formally verified, specification of the protocol, which abstracts away from details about how cryptographic and communication operations actually take place, and fill the semantic gap between formal specification and implementation without losing the verified protocol properties.

One of the limitations that still affects the methods of this kind proposed so far [12, 13] is that they do not allow the development of interoperable implementations of standard security protocols. This limitation is due to the inability of the proposed methods to set specific algorithm parameters for each cryptographic operation, and to handle arbitrary encoding/decoding schemes. In order to make such methods applicable to real protocols, the above limitations must be eliminated.

This paper shows, using a case study, how the technique originally presented in [12] has been improved in this direction. The specific approach considered starts from a Spi Calculus [1] specification of the security protocol, which can be manually derived from the protocol informal specification. The use of the Spi Calculus allows two important steps to be performed:

1. Verify that the formal specification is correct (i.e. it satisfies the intended security requirements), and thus that it does not contain any flaw itself;
2. Automatically generate the Java code that correctly implements the security protocol.

Step one, i.e. verification of the Spi Calculus protocol, can be achieved using one of the available verification tools, such as S3A [4] or Proverif [3]. Step two, i.e. automatic Java code generation, can be achieved using the code generation tool spi2java [12], and, specifically, the new version of the tool that enables the development of interoperable protocol implementations.

The case study that is presented in order to illustrate the potentiality of the proposed approach, is the development of a client for the SSH Transport Layer Protocol [15]. The main goal is to show how this approach helps a programmer to write interoperable and secure implementations of standard cryptographic protocols quickly.

The remainder of this paper is organized as follows. Section 2 introduces the last version of the spi2java tool, whose main achievement is the support for getting interoperability of the generated code. Section 3 describes the steps needed to implement the client side of the SSH Transport Layer Protocol in Java using the methodology implied by the spi2java tool. Section 4 illustrates the experiments that have been carried out to test the interoperability of the client with third party server implementations, thus assessing the fact that the developed client can be used in real environments. Moreover, some measures on the generated code and their interpretation are given. Section 5 compares the approach presented here to other existing approaches. Finally, section 6 concludes with an overview of the achievements that have been reached with the new version of spi2java, most notably the possibility to generate reliable and interoperable implementations of standard security protocols.

2 The new spi2java tool features

The spi2java tool is a software that translates a Spi Calculus definition of a security protocol into a Java program that implements it.

In order to create an interoperable Java application from the Spi Calculus source, spi2java needs to fill all the implementation details that are abstracted away by the Spi Calculus language. These implementation details can be grouped into two main categories:

1. Cryptographic and Configuration parameters

2. Encoding/decoding functions

The first group of details specifies parameters such as "what algorithm must be used for a particular encryption operation" or "what network interface must be used by a particular channel". In order to make the generated code compliant with the implemented protocol, it is necessary that these parameters can be set independently, at compile time or at run time, for each data item.

The second group of details deals with the transformation from the internal representation of messages into their external representation, and vice versa. The internal representation is the one used to perform all the operations prescribed by the protocol logic on the data; the external representation is the stream of bytes that must be exchanged with the other parties. Decoupling internal and external representations is also necessary in order to obtain interoperability, because the external representation allows the user to exactly specify, byte per byte, how data are exchanged with other actors.

The last version of spi2java improves the one described in [12] by adding the two requested features:

1. Specific implementation details can be set for each Spi Calculus term that is declared in the specification;

2. An encoding/decoding layer tailored for the protocol that is being implemented can be specified.

Another task that spi2java carries out is to statically assign a type to each Spi Calculus term. This is necessary because Spi Calculus is an untyped language, while Java is statically typed.

In order to set the specific implementation details and the statically assigned type for each Spi Calculus term, the last version of spi2java uses an eSpi (extended Spi) document, which is coupled with the original Spi Calculus source. The eSpi document is automatically generated by spi2java. The tool also fills all needed data with default values. The user can later change the proposed values to accommodate needs; after editing, spi2java checks the user-given values for correctness and coherence with the Spi Calculus specification and the eSpi document format.

Moreover, with the new spi2java version, cryptographic and configuration parameters can both be specified statically at compile time, or can be dynamically resolved at run time. The latter behaviour allows the implementation of protocols, like the SSH Transport Layer Protocol, that prescribe cryptographic algorithm negotiation at run time; the negotiated algorithm is stored into a Spi Calculus term whose value will be used as parameter for a cryptographic operation.

In order to create the encoding/decoding layer, four Java methods must be implemented by the programmer for each type of encoding/decoding that is required by the specification. They are: `encodePayload()`; `serialize()`; `decodePayload()`; `deSerialize()`.

The first method is responsible for translating the internal representation of a term into the payload, encoded as requested by the informal protocol specification. The second method is used to add the necessary headers and trailers to the payload. This approach gives high flexibility by allowing different and independent encodings for cryptographic and network operations.

The third and fourth methods are dual with respect to the first and second methods. `deSerialize()` extracts the payload from the whole data sent by the other parties, and `decodePayload()` transforms the encoded payload into the internal representation of the term.

For convenience and agile prototyping, a default encoding/decoding layer, which uses the Java serialization, is provided; however, in real environments, this default encoding/decoding layer has to be substituted with a user given one in order to implement the desired protocol.

When the Spi Calculus specification, the coupled eSpi document, and the encoding/decoding layer are done, spi2java has all the information required to generate the Java code. The Java generator engine navigates all the expressions listed in the Spi Calculus source and translates each of them into a list of semantically equivalent Java statements. The mapping between a Spi Calculus expression and its corresponding list of Java statements is called a translation rule.

In order to get high confidence about the correctness of the translation rules, spi2java comes with a Java library, *spiWrapper* (previously called *secureClasses*), that allows to map one to one Spi Calculus processes onto Java statements. This library makes the translation rules simple, and thus the latter can be manually checked with ease. Moreover, the semantics of the spiWrapper library itself is currently monitored with a test driven development.

High confidence about the Java generator engine is achieved because it only uses a small and fixed number of translation rules. For this reason, the correct behaviour of the generator can be easily checked with test beds.

3 Using spi2java to Implement the SSH Transport Layer Protocol

This section shows how a Java client for the SSH Transport Layer Protocol (SSH-TRANS) can be implemented with the help of spi2java. The implementer will have high confidence about the correctness of the security properties of the client implementation, since the Java code will be generated semi-automatically from a verified Spi Calculus specification of the protocol.

In the design phase, the Spi Calculus model of the client side of SSH-TRANS is derived from its informal specifications [14, 15, 11]. For the sake of clearness a typical SSH-TRANS scenario is provided in figure 1.

A Spi Calculus specification[1] of an SSH-TRANS client is:

```
1   sshClient(ID_C, c_algorithms) :=
            c<ID_C>.
3           c(ID_S).
            (@c_cookie)
            c<c_cookie, c_algorithms>.
6           c(KEX_S).
            let (s_cookie, s_algorithms) = KEX_S in
            let (g, p, q, DHHash, SignHash, SignKeyType, SignMode, SignPadding) =
9                   s_algorithms in
            (@x)
            c<H(g,x,p)>.
12          c(PubKey_s, f, sshash).
            case sshash of [{shash}]PubKey_s in
            [shash is H(H(ID_C, ID_S, (c_cookie, c_algorithms), KEX_S, PubKey_s,
15                  H(g,x,p), f, H(f,x,p)))]
            0
```

At line 1 the Spi Calculus process sshClient is declared with two formal parameters. ID_C represents

[1]With syntactic sugar: tuples will be reduced to nested pairs by the Spi Calculus compiler.

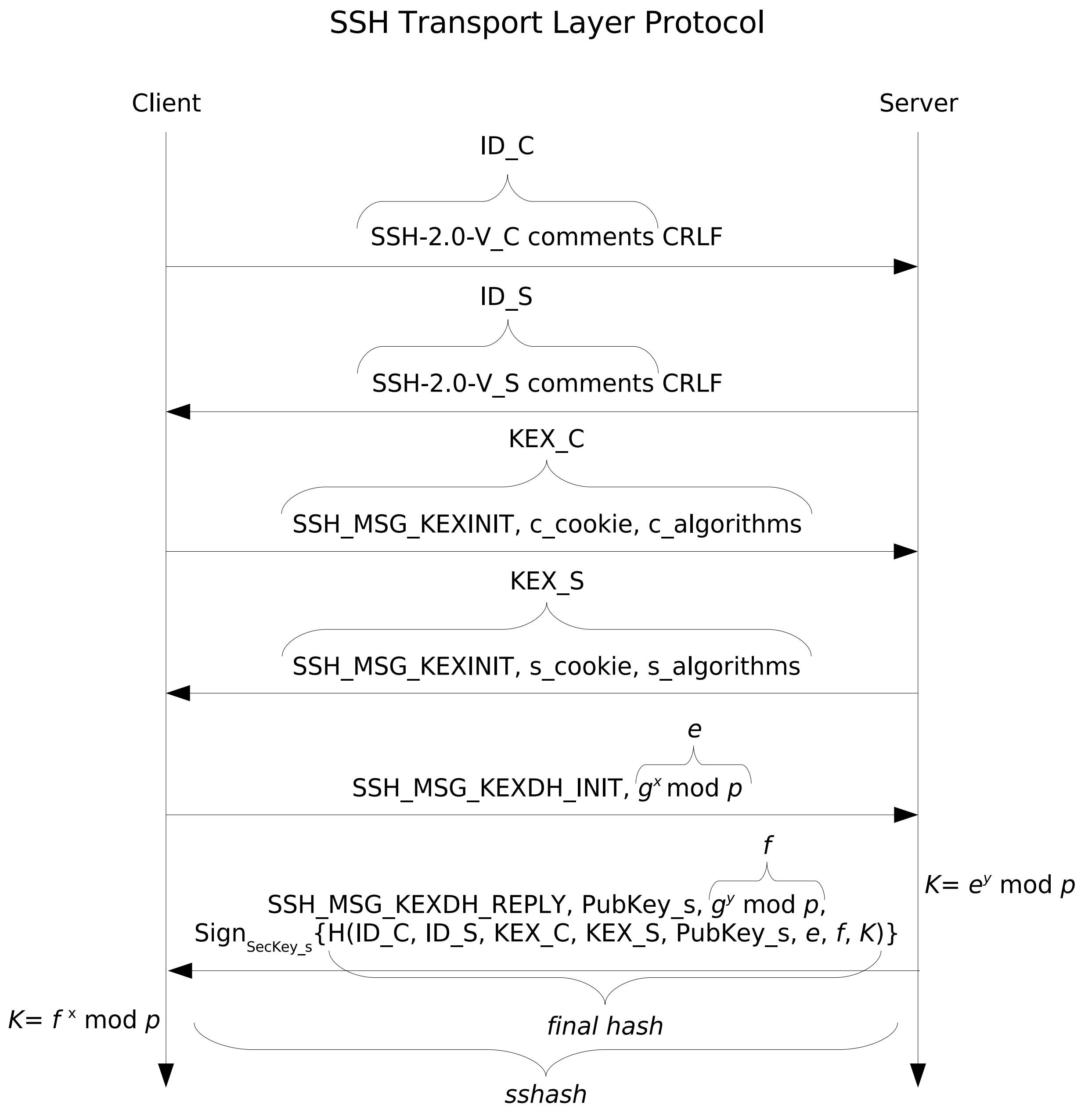

Figure 1. SSH Transport Layer Protocol typical scenario.

the client identification string and c_algorithms represents the list of algorithms supported by the client, ordered by preference. At line 2 the client sends ID_C to the server on channel c, and at line 3 it receives the server identification string ID_S from the same channel. At lines 4-5 the client sends, on channel c, the KEX_C message, that is a fresh cookie (c_cookie, generated at line 4), followed by its list of supported algorithms. Note that the message tag SSH-MSG-KEXINIT is not explicitly reported in Spi Calculus, because it is an encoding feature. At lines 6-7, the client receives the KEX_S message, that contains the server cookie s_cookie and its list of supported algorithms s_algorithms. Note that let constructs are used to split messages into their constituent parts. At lines 8-9, the client parses the server supported algorithms list, obtaining the negotiated algorithms that will be used later. At line 10 the client generates its Diffie-Hellman (DH) private key in variable x, and at line 11 it sends out its DH public key, obtained as $g^x \bmod p$. The latter has been modeled as an hash function, as explained below. Then at line 12 the client receives the server public key PubKey_s, the server DH public key f and the server signed *final hash* sshash. At lines 13-15 the client checks that the server signed *final hash* is valid against the locally computed *final hash*. If the server signature is valid the protocol ends well, and the session key, obtained as $f^x \bmod p$, and also modeled as an hash function, is established.

In order to fully understand the meaning of lines 13-15, it must be pointed out that, for asymmetric key encryption, the Spi Calculus language only offers cryptographic primitives to cipher/decipher payloads. In particular, the Spi Calculus process of *signature check*, despite of its name, does not check that a signature is valid, rather it is a mere decipher operation. Since the SSH-TRANS protocol prescribes to use RSA with SHA-1 [9] as signature algorithm, the server sends to the client the *final hash*, hashed with SHA-1, then ciphered with its private key. At line 13 the client deciphers the server signature and obtains its SHA-1-hashed *final hash* (shash), then at lines 14-15 it compares the locally generated *final hash*, hashed with SHA-1, with the received one.

It is also worth to note how the modular power operations required by the DH algorithm are handled. Spi Calculus does not offer an explicit term for modular power, however this operation and hash functions share the property of being *non-invertible*, so it is possible to model the modular power as an hash. This model implies that the Spi Calculus specification does not represent all the other properties which are particular to the modular power and not present in a cryptographic hash function. Adding the modular power operation to the Spi Calculus language may solve this issue, thus enabling a more precise analysis of the protocol.

At lines 8-9 the terms `DHHash`, `SignHash`, `SignKeyType`, `SignMode` and `SignPadding` will contain the negotiated parameters, respectively for the DH algorithm and for the signature algorithm. It can be noticed that these terms are not explicitly referenced anymore in the Spi Calculus source code. However, they are needed. These terms will be referenced in the eSpi document as the value of *variable* (run time) parameters of other terms, like the abstract hash operations denoted by `H()`.

This SSH-TRANS client only supports the RSA signature algorithm. The support of both RSA and DSA algorithms would increase the complexity of the Spi Calculus code, without adding value to the case study. Indeed, the negotiation algorithm is client driven, that is, the server will agree on the signature algorithm suggested by the client. It follows that as long as the client will require the RSA algorithm, this algorithm will always be used.

When the Spi Calculus specification is done, spi2java is used to automatically generate the coupled eSpi document, which comes filled with default values. Term types, and cryptographic and configuration parameters, are then modified to suite the protocol specification; by now the default encoding/decoding layer is used. It is worth to mention three significant changes to the default values that are needed for this protocol:

1. the type of terms `H(g,x,p)` and `H(f,x,p)` is set to "DH Hashing", and not to "Cryptographic Hashing". The DH Hashing type implements the modular power operation, as previously discussed.

2. the term `DHHash` is referenced as the *variable* parameter that contains the hash algorithm name for the *final hash*.

3. the terms `SignHash`, `SignKeyType`, `SignMode` and `SignPadding` are referenced as the *variable* parameters that contain the algorithm parameters for the server signature check, which is composed of the decryption at line 13, followed by the most external hash at lines 14-15.

When all required changes are performed, spi2java is run again in order to check that the custom eSpi document is valid and coherent.

The last step that must be accomplished before the automatic generation of the Java code, is to write an encoding/decoding layer for the SSH-TRANS, and then update the eSpi document to use this new encoding/decoding layer.

The SSH binary protocol, described in [14], specifies the binary packet format, and how each SSH type must be encoded. A Java library has been implemented, in order to translate each SSH type encoding into the internal SpiWrapper type representation, and vice versa. Using this library, the real encoding/decoding layer is a set of Java classes, where each class generates the SSH binary packets starting from the internal representations of the corresponding SpiWrapper types, or parses a received SSH binary packet and stores it into the internal representation of the corresponding SpiWrapper type.

The Java class that decodes the `s_is` term must be described separately, because it is responsible for both parsing the server algorithms list and also negotiating which algorithms will be used between client and server. When the server message is received it is not merely stored in the internal representation of the `s_is` term for later use, instead, the algorithm described in [15] is followed, and the server algorithms list is matched against the client algorithms list, and only the agreed algorithms are stored into the internal representation for later use.

When the encoding/decoding layer is done, and the eSpi document is updated to use it, spi2java is used to generate the Java code that implements the SSH-TRANS client.

Before running the client, the Spi Calculus process input arguments must be initialized.

The first parameter is the client identification string. Since it is a parameter, the programmer can change the client name and version number without changing the Spi Calculus specification.

The second parameter is the client list of supported algorithms. The order of each algorithm in the list is important, because it will influence what algorithm will be used between client and server during communication. Since this list is an input parameter, it can be modified at compile time, or even created at run time upon user preference, without the need to modify the Spi Calculus source code.

4 Experimental results

The generated SSH-TRANS client has been tested against six third party server implementations; five kinds of experiments have been executed with each server, totalizing thirty experiments. Since the negotiated algorithms depend on client preferences, the client lists of preferred algorithms have been properly configured, such that, for each kind of experiment, different algorithms would have been negotiated.

Table 1 shows, for each kind of experiment, the lists of preferred algorithms that the client sends to the server.

Kind of Experiment	Signature	DH group	Final Hash
1	RSA; DSA	1; 14	SHA-1
2	RSA; DSA	14; 1	SHA-1
3	DSA; RSA	1; 14	SHA-1
4	RSA; DSA	1	SHA-1
5	RSA; DSA	14	SHA-1

Table 1. Lists of preferred algorithms

In experiments of kind 1, 2, and 3, the client sends to the server a list with all the algorithms that the SSH-TRANS requires to be supported by the actors. If the server supports at least one of the algorithms proposed by the client, then the negotiation algorithm is expected to terminate with success, and experiments of kind 1 and 2 must end well. Instead, experiments of kind 3 are expected to correctly negotiate the algorithms, but then the client is expected to fail, due to unsupported signature algorithm.

In experiments of kind 4 and 5, for “DH group”, the client sends to the server a list with only one group. The negotiation algorithm is expected to fail if the server does not exactly support the client requested group, otherwise the experiment must end well.

The third party servers used for testing are reported, with some comments, in table 2.

Servers with comment “All correct” support all required algorithms, and all kinds of experiments end as expected. In particular, the negotiated algorithms are always the preferred client algorithms.

Servers with comment “No DH group 14” only support one of the two requested DH groups, that is group 1.

Server	Comments
OpenSSH_4.2p1	All correct
PragmaFortress 4.0	All correct
cryptlib (KpyM 1.13)	No DH group 14
lshd-2.0.2	All correct
dropbear_0.48	No DH group 14
3.2.9.1 SSH Secure Shell	No DH group 14

Table 2. Tested servers

With these servers, experiments of kind 1 and 3 end as expected. Experiments of kind 2 end correctly, however, due to the fall-back mechanism of the SSH-TRANS negotiation algorithm, the DH group 1, instead of group 14, is agreed between client and server. Experiments of kind 4 end as expected, while the ones of kind 5 correctly fail, because it is impossible to agree on a DH group.

The experiments illustrated here show the fact that the generated client can execute in real environments, because it is able to correctly interoperate with third party implementation servers.

Moreover, the client has been tested against the SSHredder[2] suite, composed of more than 650 incorrect protocol sessions. Each incorrect session has been correctly rejected by the client, which confirms the reliability of the code developed with the proposed method.

Finally, some measures of the client code are reported in table 3.

Package	TLOC[a]	MLOC[b]
spiWrapper	2064	1267
spiWrapperSSH	1557	733
sshClient	218	169

[a]Total Lines of Code: non-blank and non-comment lines in a class.

[b]Method Lines of Code: non-blank and non-comment lines inside method bodies of a class

Table 3. Measures of the generated code

The three Java packages are organized as follows:

[2]http://www.rapid7.com/securitycenter/sshredder.jsp

spiWrapper contains the *spiWrapper* library, which is used to implement the operations prescribed by the protocol logic. The code inside this package is shipped with the spi2java tool.

spiWrapperSSH contains the encoding/decoding layer for the SSH protocol. The code inside this package is manually written by the programmer that uses spi2java to implement the SSH-TRANS client.

sshClient contains the protocol logic. All the code inside this package is automatically generated by spi2java, the only exceptions are the protocol parameters, which have to be manually set by the programmer.

It can be argued that *TLOC* and *MLOC* metrics highly depend on coding style. This is true; however, both spiWrapper and spiWrapperSSH packages have been written by the same author and with the same coding style. Because of this, it can be assumed that, in this particular environment, both *TLOC* and *MLOC* are significant.

As it can be clearly noticed from the *TLOC* and *MLOC* metrics, the code that must be manually written by the programmer, that is the code inside the spiWrapperSSH package, is less than half of the whole code required by the entire application. This measure allows to state that the spi2java tool helps to make application development quicker and safer, because less code must be manually written, and the provided code has high confidence about its correctness.

5 Related Work

Some work has recently been done aimed at ensuring correctness of security protocol implementations.

The approaches proposed in [2] and in [10] are the dual of the approach presented here. They extract a verifiable formal model from an interoperable implementation of a security protocol. The model extraction approach has the advantage of allowing existing implementations to be verified without changing the way applications are currently written. However, these methods either put constraints on the syntax of the accepted source code, indeed not allowing to verify existing software, or they extract an approximate model where low level issues, like buffer overflows, are not checked, because abstracted away, and the over approximation can cause false alarms. Moreover, these approaches expose the programmer, all at once, to the whole protocol logic and implementation complexity, because a complete protocol implementation must be provided, before it can be verified.

An approach very close to the one presented in the previous version of spi2java is described in [13]. However, the generated code is neither interoperable, nor algorithms and parameters can be selected at runtime or for a specific cryptographic operation. Moreover it uses a modified version of the Spi Calculus, not allowing the formal model to be verified with existing tools that accept the plain Spi Calculus.

ACG-C# [8] is a tool that automatically generates C# code from a verified Casper script. This tool does not deal with the interoperability of the generated code. Moreover, the workflow is error prone, because it is necessary to manually modify the verified Casper script in order to let ACG-C# generate the code. It is also worth to note that the Casper scripting language is not as expressive as the Spi Calculus language. Because of this, with ACG-C# it is difficult to exactly model the behaviour of actors, as prescribed by the informal protocol specification.

The work presented in [5] illustrates another approach useful to translate formal specifications into Java code. In particular, the problem of mapping abstract data types into implemented Java classes is addressed in [6]. It is proposed to represent each allowed abstract data item as a byte array. However, this work does not take into account interoperability, because it does not deal with the different encodings (different byte array representations) that can be assumed by the same abstract data item, when it is sent to, or received from other actors.

In [7], a formal model of a security protocol used by smart cards is manually derived and refined from informal specifications, then a manual Java implementation of the refined model is provided. Finally, JML properties are manually added to the Java source. No automatic tools are used to refine the model, nor to generate the Java implementation and the JML properties. This approach is error prone, because it requires manual work in all development stages. However, this solution is still interesting, because it leads to Java code that can be directly verified.

6 Conclusions

The original work presented in this paper shows how the last version of the spi2java tool overcomes previous issues, allowing, for the first time, to semi-automatically generate, from a formal specification of a security protocol, interoperable Java code, with a high confidence about its correctness.

By providing a consistent development framework, spi2java enables other developers to reproduce the innovative results obtained here, and to achieve new ones.

The proposed case study on the SSH-TRANS client is also original work. At the same time it shows the new features of the last version of the spi2java tool, and, up to our knowledge, it is the first semi-automatically generated interoperable Java software that is developed with the proposed framework, and that works in real environments.

In order to achieve these results, spi2java has been revised, and new features have been added. However, future work is still possible. In particular, automatic generation of the encoding/decoding layer would speed up the development process, and would further improve the assurance level about the correctness of the implementation.

Although it is likely that some parts of spi2java can still be improved, the achievements showed in this paper represent innovative results, that allow programmers to develop, in less time, better interoperable security software, by using the reliability given by formal methods, the speed up given by automatic code generation, and the flexibility offered by the last version of spi2java.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[2] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop*, pages 139–152, 2006.

[3] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop*, pages 82–96, 2001.

[4] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Trans. Softw. Eng. Methodol*, 12(2):222–284, 2003.

[5] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing provable secure M-commerce applications. In *Emerging Trends in Information and Communication Security*, volume 3995 of *Lecture Notes in Computer Science*, pages 115–129, 2006.

[6] H. Grandy, K. Stenzel, and W. Reif. Refinement of security protocol data types to java. In *Proceedings of Password '06, Program Analysis for Security and Safety Workshop Discussion, Nantes, France*, 2006.

[7] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in java card. In *Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226, 2003.

[8] C.-W. Jeon, I.-G. Kim, and J.-Y. Choi. Automatic generation of the C# code for security protocols verified with casper/FDR. In *International Conference on Advanced Information Networking and Applications*, pages 507–510, 2005.

[9] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[10] J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *International Conference on Formal Methods and Models for Co-Design*, pages 89–98, 2005.

[11] S. Lehtinen and C. Lonvick. The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250 (Proposed Standard), Jan. 2006.

[12] D. Pozza, R. Sisto, and L. Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *International Conference on Advanced Information Networking and Applications*, pages 400–405, 2004.

[13] B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Services*, Toulouse, France, 2004.

[14] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[15] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), Jan. 2006.